Verification of Behavioral Consistency in C by Using Symbolic Simulation and Program Slicer

Takeshi Matsumoto, Thanyapat Sakunkonchak, Hiroshi Saito†, Masahiro Fujita
Department of Electronic Engineering, The University of Tokyo
† Research Center for Advanced Science and Technology, The University of Tokyo
E-mail:{matsumoto, thong}@cad.t.u-tokyo.ac.jp, hiroshi@hal.rcast.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

Abstract

In this paper, we propose a verification method to check the behavioral consistency of two given C descriptions. The main idea of the method is to reduce the verification effort by verifying only the program codes where the behavioral consistency must be checked. For this purpose, textual differences between the descriptions are identified at first. Then, based on these differences, the codes relevant to these differences are extracted by using the technique of program slicing. Then, the behavioral consistency between the descriptions is verified by applying symbolic simulation for the extracted codes. Our proposed method was tested on several examples. Because the verification effort was restricted to only the extracted codes, the behavioral consistency of the descriptions was efficiently verified.

1: Introduction

Verification to validate designs is one of the most important tasks in VLSI designs. However, because of the great advance in integration technologies, the verification for a whole design is getting more and more difficult. Identifications of design errors in later stages of designs significantly decrease design productivity because such errors require us to re-do designs from the specification-level. Therefore, verification in early stages of designs is an indispensable task.

In this work, we propose a verification method to check the behavioral consistency of two C descriptions efficiently. It is a part of our verification framework for hardware/software co-designs as shown in Figure 1. In this paper, for simplicity, C descriptions without pointers, recursive calls, and so on are considered. Since the inclusion of them makes verification hard, we are going to consider them in our future work together with the verification of other kinds of properties.

The main idea of our proposed method is to reduce the verification effort by verifying only the program codes where the behavioral consistency must be verified. For this purpose, at first, we find out textual differences in the two descriptions. It is similar to identify cut-points in the equivalence checking of combinational circuits [1]. After textual differences are identified, we extract the descriptions which are relevant to the differences. This is carried out by using a program slicer [2]. Finally, only the extracted program codes are verified in terms of symbolic simulation.

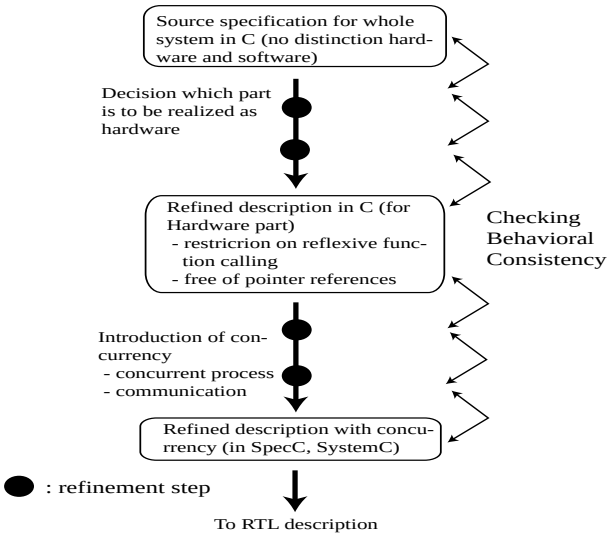

Fig. 1. Design/verification flow

This kind of consideration is very important because the ability of symbolic simulation is very sensitive to the size of descriptions. From this reason, our proposed method has a good perspective for the verification of large descriptions. In particular, it is very useful when two given descriptions are very close to each other because the number of differences to be verified is few. It is similar to the consideration in [3] where combinational equivalence checking at assembly-level is considered. However, the verification based on textual differences is very hard if there exist many differences. In that time, we may extract the descriptions not by differences but by similarities as in cut-points of combinational equivalence checking [1].

As related work, Clarke et al developed a verification method to check the behavioral consistency between a description written in Verilog hardware description language and a description written in ANSI-C by using SAT-based bounded model checking [4]. Although our method checks the behavioral consistency in terms of symbolic simulation, SAT-based verification can be used instead of symbolic simulation. Different from the method, we focus on to

reduce the verification effort by verifying only the program codes where the behavioral consistency must be checked. As a result, our proposed method significantly improve the verification time. This is particularly important for formal verification because it is very sensitive to the size of descriptions.

The organization of this paper is as follows. In section 2, we describe several basic notions used in this paper. In section 3, we explain our verification method in detail. Case studies by using our proposed method are described in section 4. Finally, we conclude this work in section 5.

2: Basic Notions

2.1: Symbolic Simulation

In this work, the behavioral consistency between two C descriptions is verified in terms of symbolic simulation. Since variables in the descriptions are treated as symbols rather than bit vectors, symbolic simulation can verify large designs efficiently.

In symbolic simulation, by checking whether variables in the two descriptions are consistent or not, the behavioral consistency between the descriptions is verified. If a variable is consistent in the two descriptions, the variable is recorded in an equivalent class (EqvClass).

In the following, we show how symbolic simulation checks the behavioral consistency between the descriptions. It is based on the method in [5].

- Symbolic simulation starts from the beginning of the two descriptions (i.e., *main* function).
- During a simulation, when an assignment is encountered, a new equivalence class (EqvClass) is created. Both left side and right side of the assignment are recorded in the EqvClass.
- For such a new EqvClass, we check whether it is consistent to previously created EqvClasses or not. It is checked by substituting the variables in the new EqvClass to the variables in the previously created EqvClasses.
- If the consistency of a variable is confirmed by the substitution, we merge the new EqvClass and the EqvClass which is consistent.
- After symbolic simulation, two variables are said to be consistent if they belong to the same EqvClass.

During symbolic simulation, the following decision procedures are considered.

1) When a function has the same name and arguments in the two descriptions, it is treated as an uninterpreted function.
2) When a case split arises because of if or $switch$ statement, only executable paths are simulated. If they cannot be identified, all paths are simulated.

If the behavioral consistency is not proved by symbolic simulation because of the existence of complex expressions, we use Stanford Validity Checker (SVC) [6] to prove the consistency of such complex expressions. Currently, we are considering how to involve SVC in our symbolic simulator.

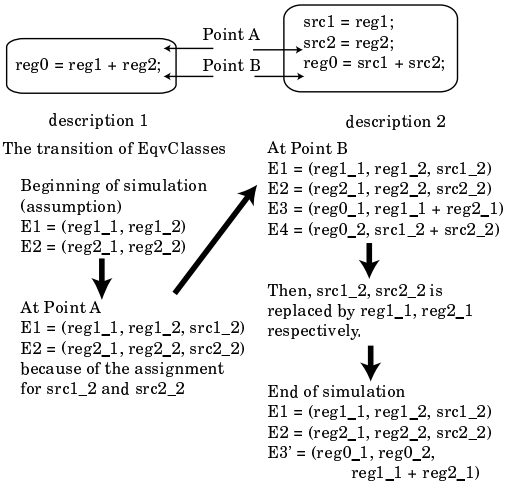

Fig. 2. An example of symbolic simulation

1) Example: Figure 2 shows an example of symbolic simulation. In this example, we verify the behavioral consistency of variable $reg0$ in the two descriptions. Note variable v is denoted as v_1 and v_2 in description 1 and description 2. For convenience, we assume that variables $reg1$ and $reg2$ are consistent in both descriptions (i.e., two EqvClasses E1 and E2 are initially created).

First of all, assignments for $src1$ and $src2$ in description 2 are simulated (i.e., point A in Figure 2). This results in that $src1\_2$ is inserted into E1 and $src2\_2$ is into E2, because these assignments say that $src1\_2$ ($src2\_2$) is equal to $reg1\_2$ ($reg2\_2$). Next, E3 and E4 are newly created before to reach point B. Namely, the assignment for $reg0$ in both descriptions is recorded into E3 and E4. After the creations of E3 and E4, $src1\_2$ and $src2\_2$ in E4 are substituted by $reg1\_1$ and $reg2\_1$, since $src1\_2$ ($src2\_2$) and $reg1\_1$ ($reg2\_1$) are in the same EqvClass E1 (E2). From this substitution, we can identify that E3 and E4 are identical (i.e., they are merged into E3'). As a result, we can conclude that $reg0$ is consistent in both descriptions.

2.2: Program Slicing

Program slicing is an operation that identifies semantically meaningful decompositions of programs based on data- and control-flow analysis [7]. For slicing, we have to decide a *slicing criterion* (i.e., variables in a program point where user has an interest). Therefore, a program slice of a variable v consists of a set of program codes that can potentially affect (or be affected by) the variable v. There are two kinds of slicing: backward slicing and forward slicing. Backward slicing for the variable v extracts all of the codes that affect the variable v (i.e., underlined codes in Figure 3 represent

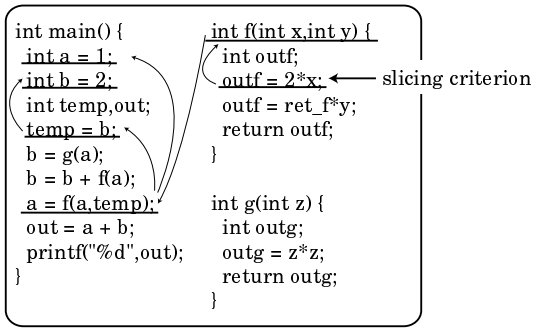

Fig. 3. An example of backward slicing

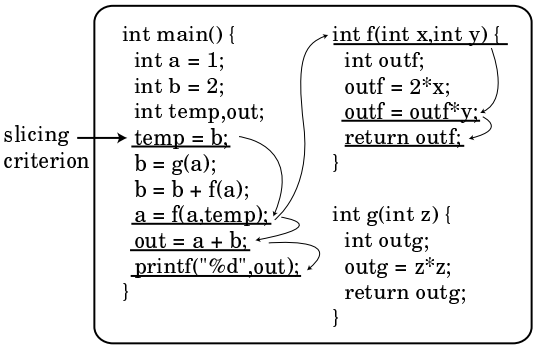

Fig. 4. An example of forward slicing

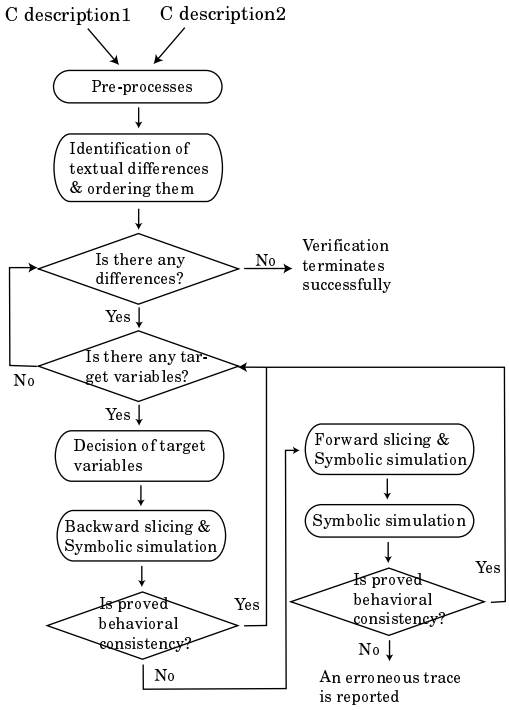

Fig. 5. Verification flow

the backward slicing of variable $outf$). On the other hand, forward slicing for the variable v extracts all of the codes that are affected by the variable v (i.e., underlined codes in Figure 4 represent the forward slicing of variable $temp$).

3: Verification Strategy

In this section, our verification strategy to check the behavioral consistency of two given C descriptions is described. Figure 5 shows a verification flow which captures our verification strategy.

3.1: Pre-processes

At first, pre-processes such as in-lines of macro definitions and removals of comments are carried out. In addition, we remove all of the standard input/output operations such as $printf()$ and $puts()$. This is because we are focusing on only the verification of functionalities such as variable assignments.

Then, for each description, a *system dependency graph* (SDG) [2] which captures the flow of control, data, and function calling in the description is constructed. Since we simulate the two descriptions in the order of program executions, such an internal representation is required. In other words, we cannot identify the execution order from the descriptions directly. Basically, program slicers internally construct an SDG to analyze descriptions. Therefore, the information required for the verification can be derived from the SDG.

3.2: Identification of Textual Differences

Next, we identify the textual differences between the descriptions by using a UNIX standard command "diff". These identified textual differences give us hints about the places where the occurrences of inconsistent assignments are. Such differences are used as slicing criteria for the slicing of the two descriptions. To take meaningful differences, we must arrange the descriptions so that both descriptions have the same coding style. This is because "diff" just extracts textual differences, the differences of coding styles (e.g., positions of spaces, etc.) are also identified although they are irrelevant for the verification of behavioral consistency. After taking differences, we sort them in the order of program executions by referring to the SDG, since symbolic simulation verifies these differences in the order of program executions.

For example in Figure 6, there exist three differences between description 1 and description 2. Since the execution of the descriptions starts from the top of them, we denote these differences as $d1$, $d2$, and $d3$ as in Figure 6. The behavioral consistency of variables in $d1$ is verified at first.

3.3: Verification based on Symbolic Simulation and Program Slicing

Verification based on symbolic simulation and program slicing is carried out for each identified difference. There-

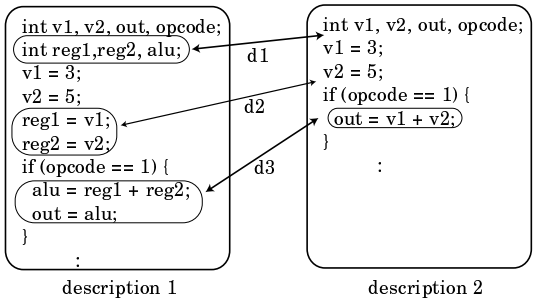

Fig. 6. Identifications of textual differences

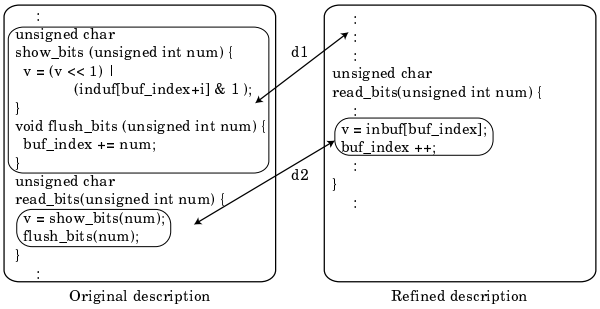

Fig. 7. The textual differences in case study 1

fore, if in all of the differences, the behavioral consistency is proved as *true*, the given descriptions are said to be consistent. On the other hand, if there exists a difference where the assignment is inconsistent, the verification terminates while producing an error information.

1) Decision of Target Variables: Since a difference may have several assignments, *target variables* to be verified are decided. Target variables are variables which are declared in both descriptions and assigned within the difference. If a variable v declared in both descriptions is assigned in only one of the descriptions in a difference d, we insert a dummy assignment $v = v$ in the other description to make a correspondence for this variable in the difference.

For example of Figure 6, there is no target variable in differences $d1$ and $d2$ because they represent the declaration and the assignment of variables internal to description 1. On the other hand, since variable out is declared in both descriptions and assigned in difference $d3$, it is a target variable. Therefore, we must check whether variable out in difference $d3$ is consistent in both descriptions.

2) Application of Backward Slicing and Symbolic Simulation: In advance to the symbolic simulation for a target variable in a difference, backward slicing is applied for the variable to extract the codes which affect the variable.

After the program codes relevant to the target variable are extracted, symbolic simulation is carried out from the beginning of the codes. Verification paths are classified into two cases according to the result of the simulation.

Case 1. Behavioral consistency is proved. If the behavioral consistency of the target variable is proved after the symbolic simulation, the next target variable is selected if it exists in the same difference.

Case 2. Behavioral consistency is not proved. On the other hand, if the symbolic simulation says that the behavioral consistency of the target variable is false or not decided, further verification is required from the point that the variable is assigned. Since the simulation with backward slicing verifies only the program codes which affect the variable, there is a possibility that the variable will be consistent in the codes which are affected by the variable. From this reason, forward slicing is carried out to extract the codes which are affected by the variable.

If the behavioral consistency of the variable is proved in the extracted codes, the next target variable is selected. Otherwise, an error information is reported. Since symbolic simulation keeps all of the simulation traces by means of EqvClasses, the source of inconsistent assignments is detected as an error information.

3) Avoiding redundant verifications: According to our method, there is a possibility that a portion of program codes relevant to a target variable v is already sliced and simulated. As a result, a redundant simulation may occur if such simulated codes are extracted again. To solve this problem, only the portion which is not simulated so far is verified.

4: Case Studies

Currently, in our verification method, symbolic simulation to check the behavioral consistency is implemented as a tool set. Although as a program slicer CodeSurfer [2] which is a product of GrammaTech Inc. is used, it is not integrated in our tool set yet.

As mentioned in section 1, currently our verification method can verify only C descriptions without pointers, recursive calls, and so on. Therefore, in this section, we describe two case studies which satisfy such restrictions.

4.1: Case Study 1: Function In-lining of Huffman decoder

In this experiment, we prepared the C description of Huffman decoder and the refined one after some functions are in-lined. As shown in Figure 7, the definitions of functions $show\_bits(num)$ and $flash\_bits(num)$ in the original description were in-lined in the refined one (see difference $d2$). In this case, two differences were identified and two target variables were verified. Since both descriptions were very close to each other, only 42% of the original description and 27% of the refined one were extracted for the verification (see Table I). As a result, the behavioral consistency of these two variables was proved successfully by symbolic simulation.

TABLE I

The result of case study 1

	Total lines	Extracted lines
Original	49	21
Refined	41	11
2 differences, 2 target variables		

TABLE II

The result of case study 2

	Total lines	Extracted lines
Original	632	131
Refined	630	129
6 differences, 6 target variables		

4.2: Case Study 2: MAXSAT

To check the applicability of our proposed method for relatively large descriptions, C description to calculate MAXimum SATisfiability (MAXSAT) was verified. In this experiment, a greedy randomized adaptive search procedure (GRASP) [8] which is a heuristic of MAXSAT problem was used as the original description. To obtain a refined one, we arbitrary inserted six differences in the original description so that the behaviors of both descriptions were consistent.

In both descriptions, only 20% of the description was extracted for the verification of these six differences. This result showed that our verification method was very useful for the verification of relatively large descriptions.

5: Conclusions and Future Work

In this paper, we proposed a verification method to check the behavioral consistency between two given C descriptions. The main feature of our proposed method is to reduce the verification effort by verifying only the program codes where the behavioral consistency must be checked. For this purpose, textual differences between descriptions are identified at first. Then, the program codes relevant to such differences are extracted by the technique of program slicing. The extracted codes are finally verified in terms of symbolic simulation. Since symbolic simulation is very sensitive to the size of program codes, our proposed method can verify large descriptions efficiently. In particular, it is very useful when two given descriptions are very close to each other. This fact was confirmed by two case studies.

As future work, we are going to complete the tool implementation. In addition, we are going to extend the method so that it can handle descriptions with pointers, recursive calls, and so on. Furthermore, by using the code extraction method proposed in this paper, we are going to consider the verification of other properties rather than the behavioral consistency between descriptions.

References

[1] W. Donath, H. Ofek, “Automatic Identification of Equivalence Points for Boolean Logic Verification”, *IBM Technical Disclosure Bulletin*, vol. 18, no. 8, pp.2700–2703, 1976.

[2] CodeSurfer: http://www.grammatech.com/products/codesurfer/

[3] D. W. Currie, A. J. Hu, S. Rajan, M. Fujita, “Automatic Formal Verification of DSP Software”, *Proc. Design Automation Conference*, pp.130–135, Jun. 2000.

[4] E. M. Clarke and D. Kroening, “Hardware Verification Using ANSI-C Programs as a Reference”, *Proc. Asia South Pacific Design Automation Conference*, pp.308–311, 2003.

[5] G. Ritter, “Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation”, PhD thesis, Darmastadt University of Technology and Universite Joseph Fourier, 2000.

[6] C. W. Barrett, D. L. Dill, J. R. Levitt, “Validity Checking for Combinations of Theories with Equality”, *Proc. Internatioal Conference on Formal Methods in Computer-Aided Design*, pp.187–201, Nov. 1996.

[7] M. Weiser, “Program slices: Formal, psychological, and practical investigations of an automatic program abstraction”, PhD thesis, University of Michigan, 1979.

[8] M. G. C. Resende, L. S. Pitsoulis, P. M. Pardalos, “Fortran Subroutines for Computing Approximate Solutions of Weighted MAXSAT Problems using GRASP”, *tech. rep.*, AT&T Research, Murray Hill, NJ, 1998.